Black-box Unit Testing (and Demo)

Team Warriors (lx2180, wp2213, rj2394, sv2525)

Class Diagrams

Class Vendor
id:integer email:string hashed_password: string registered_on: datetime

<u>Static methods:</u>
get_vendor_by_id(id): Vendor
get_vendor_by_email(email): Vendor
 	
signup_vendor(email, password)
login_vendor(email, password): Token (if succeed)
logout_vendor(token)

create_auth_token(): Token add_new_post(): Post

Class Token
id: integer token: string blacklisted_on: datetime
check_blacklisted(): True / False

Class Post
id: integer location: string latitude: float longitude: float schedule: text menu: text (optional) posted_on: datetime vendor_id: integer
<u>Static Methods:</u> get_latest_post_from_vendor(vendor_id): Post get_latest_posts_by_distance(latitude, longitude, distance): list of Post

Black-box Unit Test cases

Class Vendor:

- get_vendor_by_id(id):
  Look up the vendor by vendor id
  - Equivalence partitions: non-integers, integers <= 0, integers > 0 but not existing in database, a valid id
  - Boundary condition
    - id = 'randomstring' → lookup failure
    - id = -1 → lookup failure
    - id = 0 → lookup failure
    - a valid id that corresponds to an existing vendor → look that that vendor
    - an id that is not existent in the database → lookup failure

- get_vendor_by_email(email):
  Look up the vendor by email address
  - Equivalence partition - a valid email address in database, a valid email address not in the database, email address with invalid syntax (without @)
  - Boundary condition:
    - valid_user@ase.com → look up the vendor
    - no_such_user@ase.com → failure
    - smithatgolddotcom → failure
    - !joedon@ase → failure

- signup_vendor(email, password)
  To register a new vendor
  - Equivalence partition - email address with invalid syntax, email address already in database, valid and new email address
  - Boundary condition -
    - (smithatgolddotcom, xxxx) → failure
    - (old_user@ase.com, xxxx) → failure (already exist)
    - (new_user@ase.com, xxxx) → success
  - Output at success - Vendor will be saved in database

- login_vendor(email, password):
  To login a vendor and return an authorization token if succeed
  - Equivalence partition - email address with invalid syntax, email address not in database, password that does not match email in database, email address and password pair that both match in database
  - Boundary condition -
    - (smithatgolddotcom, xxxx) → failure
    - (no_such_user@ase.com, xxxx) → failure
    - (valid_user@ase.com, wrong_password) → failure
    - (valid_user@ase.com, correct_password) → success
  - Output at success: assigns a token.
    Output at failure: will not generate token is login is unsuccessful.

- logout_vendor(token)
  To log out the vendor, i.e. blacklisting the token assigned to the vendor
  - Equivalence partition - token with invalid syntax, a valid token but already expired, a valid and not-expired token
  - If vendor time of logging out is lesser than the time out of the token, then the system will log them out.
  - Boundary condition:
    - "random string" → failure
    - Valid token with expiration time < current time → failure
    - Valid token with expiration time > current time → success
  - Output at success: the application have automatically signed the vendor out and puts the token on the blacklist so that further access using the token is denied.

- create_auth_token():
  To create a token for each vendor at login. A token specifies the duration for which a vendor can get access to the posting site and their authorization policies.

Class Token:

- check_blacklisted():
  Checks if the token is already been blacklisted.

Class Post:

- get_latest_post_from_vendor(vendor_id):
  To filter and pull the last posting a vendor has made by the vendor id.
  - Equivalence partition: non-integers, integers <= 0, integers > 0 but not existing in database, a valid id
  - Boundary condition
    - id = -1, 0, "randomstring" → failure
    - an id that is not existent in the database → failure
    - a valid id that corresponds to an existing vendor → success
  - Output at success: return the latest post of the vendor (post contains vendor_id, location, latitude, longitude, and other information)

- get_latest_posts_by_distance(latitude, longitude, distance):
  To filter a list of latest posts which are within the required distance of the location.
  - Equivalence partition: latitude/longitude/distance not floating point numbers, distance <= 0, distance>max_distance, latitude/longitude out of range, valid tuple
  - Boundary conditions:
    - (40.80, -73.96, 1.5) → success (within 1.5mile of Columbia University)
    - (100, xx, xx) → failure (latitude out of range)
    - (xx, -200, xx) → failure (longitude out of range)
    - (xx, xx, -10) → failure (distance out of range)
    - (xx, xx, 0) → failure (distance out of range)

Frontend UI Tests
Customer

- Homepage Loading Test
  Loads the homepage of Go Trucks application as a customer (without signing in).
  Passing Condition: server returns 200 and the page completes rendering all elements.
- Vendor Listings Test
  Loads a list of nearby vendors.
  Passing Condition: the table containing a list of vendors is rendered successfully.
- Google Map Rendering Test
  Loads Google Map with vendor's location pins.
  Passing Condition: Google Map is loaded and its elements and stylings can be detected successfully

Vendor

- Vendor Page Loading Test
  Tests that the login and registration page shows up if the user selects vendor portal: the page should contain login and registration forms.
- Attempt to login:
  Login button works.
- Attempt to register:
  Registration works (prompt on success/failure).
- Customer homepage loads:
  Vendors are listed in a table; a Google Map is displayed with markers.

Implementation

Backend

(Ruoxin Jiang & Shreya Vaidyanathan)
Unit tests: https://github.com/ase-warriors/go-trucks/tree/master/backend/tests
Command to run tests: *python backend/manager.py test*

The unit testing framework we used for backend is PyUnit (i.e. the unittest module). It supports test automation and sharing of the setup and teardown code for tests. While implementing unit testing, we encountered the following challenges:

1) Test database operations: we want to be able to test database operations without interfering with the existing database for development/production. Thus, we create a separate database for testing and automate its setup/teardown in the testing process.

2) Test interaction with frontend: we also need to make sure that the backend passes the correct http responses for frontend requests. To test the integration, we use *flask-testing* which is able to simulate http requests.

We currently have 19 unit tests for backend which cover most of the equivalent partitions described in the previous section.

Frontend

(Andy Xu & Wendy Pan)
Unit tests: https://github.com/ase-warriors/go-trucks/tree/frontend/frontend/test

On the frontend side, we use three major libraries to conduct unit tests. We are using mocha.js as the unit test framework that runs on node.js, nightmare.js as the headless browser testing framework, and chai, the assertion library. Without the need to boot the browser (Chrome or Firefox), the unit test suite can conduct unit tests automatically, collect success information and failed exceptions and report the statistics.

Command to run tests: `npm test`

Sample Output of the UI unit test suite.

```
Test Go Trucks as a vendor:
  ✓ should load the vendor page (1007ms)
  ✓ should contain the login form (1015ms)
  ✓ should contain the registration form (1023ms)
  ✓ attempt to register (1145ms)
  ✓ attempt to login to the vendor page (1158ms)

 Testing Go Trucks as a customer
  ✓ should load the homepage (922ms)
  ✓ should list nearby vendors in a table (899ms)
  ✓ should render a list of vendors in Google Map (1200ms)
```

The most challenging part of frontend UI testing is to simulate user input using nightmare.js. For some unknown reasons, keystroke simulation needs to be close to real life scenarios as much as possible. In another word, the headless browser engine (react-framework) misses keystrokes in the typing speed is too fast. We had do adjust the waiting time to make the test pass.

Another challenge that we face is to use promise library in front-end testing framework. Asynchronous events requires very careful handling in case of special events that needed attentions (window.alert(), for example). We had to investigate why we were having unhandled events, which results in testcase failures (missing done() in a thread, which leads to handler timeouts.)

Demo

- What we demonstrated:

1. Customer view of the web app: Map with markers that indicate where the vendors are and a table with the most recent postings from each vendor.
2. Vendor portal: registration and login page.
3. Create posting: autocomplete when addresses are entered and prediction of exact coordinates based on the complete address.
4. Submit multiple postings.
5. Session persists on refresh of the page.

- Problems we faced:
    1. We did not account for the case where the vendor enters an invalid address.
- Recommendations from (TA) Kilol:

  1. Show current location for each vendor- update latest only- stretch goal: add a textbox for notes for vendors about regular timings in various locations (today's specials etc)
  2. Test cases - log in, log out
  3. Vendors shouldn't give wrong address and lat, long. Should validate the address so that an invalid address doesn't go through.
  4. Add a vendor name with a login - show which vendor is currently logged in.
  5. How to go about UI testing ? Chrome automation testing perhaps. Check on tools which go through and validate all the components in the page. [Which we've incorporated subsequently itself]